Coffman deadlocks in SCOOP*

Georgiana Caltais[1] and Bertrand Meyer[1,2,3]

[1] Department of Computer Science, ETH Zürich, Switzerland
[2] Eiffel Software, Santa Barbara
[3] NRU ITMO, Saint Petersburg

Abstract

In this paper we address the deadlock detection problem in the context of SCOOP – an OO-programming model for concurrency, recently formalized in Maude. We present the integration of a deadlock detection mechanism on top of the aforementioned formalization and analyze how an abstract semantics of SCOOP based on a notion of "may alias expressions" can contribute to improving the deadlock detection procedure.

Introduction. In this paper we are targeting SCOOP [8] – a concurrency model recently provided with a formalization based on Rewriting Logic (RL) [7], which is "executable" and straightforwardly implementable in the programming language Maude. *Our aim* is to develop a (Coffman) deadlock [2] detection mechanism for SCOOP applications. Intuitively, such deadlocks occur whenever two or more executing threads are each waiting for the other to finish.

Our contribution. We present the integration of a deadlock detection mechanism on top of the formalization in [8]. We also briefly analyze how a simplified, abstract semantics of SCOOP based on a notion of "may alias" expressions [4, 1] can be exploited in order to improve the deadlock detection procedure.

The literature on using static analysis [5] and abstracting techniques for (related) concurrency models is considerable. We refer, for instance, to the recent work in [3] that introduces a framework for detecting deadlocks by identifying circular dependencies in the (finite state) model of so-called contracts that abstract methods in an OO-language. The integration of a deadlock analyzer in SCOOP on top of Maude is an orthogonal approach that belongs to a more ambitious goal, namely the construction of a RL-based toolbox for SCOOP programs including a *may alias* analyzer, as thoroughly presented in [1] and a type checker.

Deadlock detection in SCOOP. The key idea of SCOOP is to associate to each object a processor, or *handler* (that can be a CPU, or it can also be implemented in software, as a process or thread). In SCOOP terminology, objects that can run on different processors are *separate* from each other. Assume a processor p that performs a call $o.f(a_1, a_2, \ldots)$ on an object o. If o is declared as "separate", then p sends a request for executing $f(a_1, a_2, \ldots)$ to q – the handler of o (note that p and q can coincide). Meanwhile, p can continue. Moreover, assume that $a_1, a_2, \ldots$ are of "separate" types. In the SCOOP semantics, the application of the call $f(\ldots)$ will *wait* until it has been able to *lock* all the separate objects associated to $a_1, a_2, \ldots$. This mechanism guarantees exclusive access to these objects. Processors communicate via *channels*.

In the context of SCOOP, the *deadlocking problem* reduces to identifying whether a set of processors reserve each other circularly. This situation might occur, for instance, in a Dining Philosophers scenario, where both philosophers and forks are objects residing on their own processors. Given a processor p, by $W(p)$ we denote the set of processors p *waits* to release the resources p needs for its asynchronous execution. Orthogonally, by $H(p)$ we represent the set of

resources (more precisely, resource handlers that) p already acquired. We say that a deadlock exists if for some set D of processors: $(\forall p \in D).(\exists p' \in D).(p \neq p') \wedge (W(p) \cap H(p') = \emptyset)$ ($\clubsuit$).

The semantics of SCOOP in [8] is defined over tuples of shape $\langle p_1 :: St_1 \mid \ldots \mid p_n :: St_n, \sigma \rangle$ where, p_i denotes a processor (for $i \in \{1, \ldots, n\}$), St_i is the call stack of p_i and σ is the *state* of the system. States hold information about the *heap* (which is a mapping of references to objects) and the *store* (which includes formal arguments, local variables, *etc.*). Integrating the deadlock definition in ($\clubsuit$) on top of the Maude formalization in [8] is almost straightforward. Given a processor p' as in ($\clubsuit$), the set $H(p')$ corresponds, based on [8], to $\sigma.rq\_locks(p')$. Whenever the top of the instruction stack of a processor p is of shape $lock(\{q_i, \ldots, q_n\})$, we say that the wait set $W(p)$ is the set of processors $\{q_1, \ldots, q_n\}$. Hence, assuming a predefined system configuration $\langle deadlock \rangle$, the SCOOP transition rule in Maude corresponding to ($\clubsuit$) can be written as:

$$\frac{(\exists D \subseteq \sigma.procs).(\forall p \in D).(\exists p' \in D).(p \neq p')\wedge}{(aqs := \ldots \mid p :: lock(\{q_i, \ldots\}); St \mid \ldots) \ \wedge \ (\sigma.rq\_locks(p').has(q_i))}{\langle aqs, \sigma \rangle \rightarrow \langle deadlock \rangle} \tag{1}$$

It is intuitive to guess that $\sigma.procs$ in (1) returns the set of processors in the system, whereas aqs stands for the list of these processors and their instruction stacks (separated by the associative & commutative operator "$\mid$"). We use "$\ldots$" to represent an arbitrary sequence of processors and processor stacks.

Discussion. We implemented (1) and tested the deadlock detection mechanism on top of the formalization in [8] for the Dining Philosophers problem. A case study considering two philosophers can be run by downloading the SCOOP formalization at:
`https://dl.dropboxusercontent.com/u/1356725/SCOOP-NWPT-14.zip`, and executing the command
`> maude SCOOP.maude ..\examples\dining-philosophers-example.maude`. In our example, the philosophers `p1` and `p2` can reach a (Coffman) deadlock (`go_wrong(p1, p2)`) if they adopt a wrong eating strategy (`pi.eat_wrong`, with $i \in \{1, 2\}$). This might happen whenever a philosopher proceeds by picking up the forks `f1` and `f2` on the table in turn (`pick_in_turn(fi)`, with $i \in \{1, 2\}$) instead of picking them at the same time. It might be the case that, for instance, `p1` picks up `f1`, whereas `p2` immediately picks up `f2`. Thus, each of the two philosophers is holding a fork the other philosopher is waiting for.

As can be seen from the code in `dining-philosophers-example.maude`, in order to implement our applications in Maude, we use intermediate representations. For a brief example, consider the class implementing the *philosopher* concept, given below:

```
(class 'PHILOSOPHER
    create { 'make } (
        attribute {'ANY} 'left : [!,T,'FORK] ; attribute {'ANY} 'right : [!,T,'FORK] ;

        procedure { 'ANY } 'make ( 'fl : [!,T,'FORK] ; 'fr : [!,T,'FORK] ; )
            do  ( assign ('left, 'fl) ; assign ('right, 'fr) ; )
            [...]
        end ;
[...] end)
```

it declares two forks − `'left` and `'right` of type `[!, T, 'FORK]`, that can be handled by any processor (`T`) and that cannot be `Void` (`!`). The corresponding constructor `'make('fl, 'fr)` initializes the philosopher's forks accordingly.

It is worth pointing out that in the aforementioned example we use a predefined strategy [6] that guides the rewriting of the Maude rules formalizing SCOOP towards a $\langle deadlock \rangle$ system configuration. Nevertheless, such an approach requires lots of ingeniousness and, moreover, is not automated. Given the size of the current SCOOP formalization, running the

Maude model checker is, unfortunately, not an option. We anticipate a "way out" of the state explosion issue by exploiting the expression-based alias calculus in [4] in order to provide a simplified, abstract semantics of SCOOP. In short, the calculus in [4] identifies whether two expressions in a program *may* reference to the same object. Consider, for intuition, the code `x := y; loop x := x.next end` that assignes a linked list. The corresponding execution causes `x` to become aliased to `y.next.next. ...`, with a possibly infinite number of occurrences of the field `next`. The set of associated "may alias" expressions identified by the calculus in [4] can be equivalently written as $\{[\, \mathtt{x}, \mathtt{y.next}^k] \mid k \geq 0\}$.

The idea behind using an alias-based abstract semantics of SCOOP stems from the fact that SCOOP processors are known from object references, which may be aliased. Therefore, the SCOOP semantics could be simplified by retaining within the corresponding transition rules only the information relevant for aliasing. Consider, for instance, the assignment instruction formally specified as:

$$\frac{\text{a is fresh}}{\langle p :: t := s; St, \sigma \rangle \to \langle p :: \text{eval}(a, s); \text{wait}(a); \text{write}(t, a.data); St, \sigma \rangle} \ .$$

Intuitively, "eval(a, s)" evaluates s and puts the result on channel a, "wait(a)" enables processor p to use the evaluation result and "write$(t, a.data)$" sets the value of t to $a.data$. The abstract transition rule omits the evaluation of the right-hand side of the assignment $t := s$ and the associated message passing between channels, and updates the aliasing information in the newly added component *alias_* (consisting of a set of alias expressions) according to the calculus in [4]:

$$\frac{\cdot}{\langle p :: t := s; St, \sigma, alias_{old)} \rangle \to \langle p :: St, \sigma, alias_{new} \rangle} \ .$$

Then, the rule (1) identifying deadlocks can be naturally redefined to range over the expressions aliased with the processors p, p' and q_i, respectively. Nevertheless, observe that this approach is prone to introducing "false positives" w.r.t. the expressions that would actually become aliased at runtime; this is due to the over-approximating nature of the alias calculus in [4] that ignores conditions in conditionals and loops. Furthermore, the abstract setting enables the simplification of the SCOOP semantics by completely eliminating the rules formalizing the exception handling mechanism, for instance. We plan to closely investigate and implement this abstraction mechanism in Maude. For a survey on similar "abstracting" procedures we refer to the work in [7].

References

[1] G. Caltais. Expression-based aliasing for OO-languages. Accepted in *3rd International Workshop on Formal Techniques for Safety-Critical Systems 2014*; to appear. *CoRR*, abs/1409.7509, 2014.

[2] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[3] E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong. Deadlock analysis of concurrent objects: Theory and practice. In *Integrated Formal Methods*, 394–411, 2013.

[4] A. Kogtenkov, B. Meyer, and S. Velder. Alias and change calculi, applied to frame inference. *CoRR*, abs/1307.3189, 2013.

[5] W. Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, 1992.

[6] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a Strategy Language for Maude. In *Electr. Notes in Theor. Comp. Sci.*, 117:417–441, 2005.

[7] J. Meseguer and G. Rosu. The rewriting logic semantics project: A progress report. In *Fundamentals of Computation Theory*, 1–37, 2011.

[8] B. Morandi, M. Schill, S. Nanz, and B. Meyer. Prototyping a concurrency model. In *13th International Conference on Application of Concurrency to System Design*, 170–179, 2013.

A Dining Philosophers in SCOOP

In what follows, we provide the relevant parts of the intermediate class-based representation in `dining-philosophers-example.maude`, together with parts of the Maude output corresponding to the strategy-based execution of the example:

```
srew

(( import default

(class 'APPLICATION
    create
        { 'make }
    (
        attribute {'ANY} 'meal : [!,T,'MEAL] ;

        procedure { 'ANY } 'make (nil)
            require True
            local ( nil )
            do
                (
                create ('meal . 'make(nil)) ;
                command('Current . 'execute_wrong('meal ;)) ;
                )
            ensure True
            rescue  nil
        end ;

        procedure { 'ANY } 'execute_wrong ( 'm : [!,T,'MEAL] ;  )
            require True
            local ( nil )
            do
                (
                command('m . 'do_wrong(nil)) ;
                )
            ensure True
            rescue  nil
        end ;
    )

    invariant True
end) ;

(class 'MEAL
    create
        { 'make }
    (
        attribute {'ANY} 'p1 : [!,T,'PHILOSOPHER] ;
        attribute {'ANY} 'p2 : [!,T,'PHILOSOPHER] ;
        attribute {'ANY} 'f1 : [!,T,'FORK] ;
        attribute {'ANY} 'f2 : [!,T,'FORK] ;

        procedure { 'ANY } 'make (nil)
            require True
            local ( nil )
            do
                (
                create ('f1 . 'make(nil)) ;  create ('f2 . 'make(nil)) ;
                create ('p1 . 'make('f1 ; 'f2 ;)) ;  create ('p2 . 'make('f2 ; 'f1 ;)) ;
                )
```

```
                ensure True
                rescue  nil
        end ;

        procedure { 'ANY } 'do_wrong (nil)
                require True
                local ( nil )
                do
                    (
                    command ('Current . 'go_wrong('p1 ; 'p2 ;)) ;
                    )
                ensure True
                rescue  nil
        end ;

        procedure { 'ANY } 'go_wrong ('pa : [!,T,'PHILOSOPHER] ; 'pb : [!,T,'PHILOSOPHER] ;)
                require True
                local ( nil )
                do
                    (
                    command ('pa . 'eat_wrong(nil)) ;
                    command ('pb . 'eat_wrong(nil)) ;
                    )
                ensure True
                rescue  nil
        end ;
    )

    invariant True
end) ;

(class 'PHILOSOPHER
    create
        { 'make }
    (
        attribute {'ANY} 'left : [!,T,'FORK] ;
        attribute {'ANY} 'right : [!,T,'FORK] ;

        procedure { 'ANY } 'make ( 'fl : [!,T,'FORK] ; 'fr : [!,T,'FORK] ; )
                require True
                local ( nil )
                do
                    (
                    assign ('left, 'fl) ;
                    assign ('right, 'fr) ;
                    )
                ensure True
                rescue  nil
        end ;

        procedure { 'ANY } 'pick_two ('fa : [!,T,'FORK] ; 'fb : [!,T,'FORK] ; )
                require True
                local ( nil )
                do
                    (
                    command ('fa . 'use(nil)) ;
                    command ('fb . 'use(nil)) ;
                    )
                ensure True
                rescue  nil
        end ;
```

```
        procedure { 'ANY } 'eat_wrong (nil)
            require True
            local ( nil )
            do
                (
                command ('Current . 'pick_in_turn('left ;)) ;
                )
            ensure True
            rescue  nil
        end ;

        procedure { 'ANY } 'pick_in_turn ('f : [!,T,'FORK] ; )
            require True
            local ( nil )
            do
                (
                command ('Current . 'pick_two('f ; 'right ;)) ;
                )
            ensure True
            rescue  nil
        end ;
    )

    invariant True
end) ;

(class 'FORK
    create
        { 'make }
    (
        procedure { 'ANY } 'make (nil)
            require True
            local ( nil )
            do ( nil )
            ensure True
            rescue  nil
        end ;

        procedure { 'ANY } 'use (nil)
            require True
            local ( nil )
            do ( nil )
            ensure True
            rescue  nil
        end ;
    )

    invariant True
end) ;

) settings('APPLICATION, 'make, false, deadlock-on))
using
init ;  parallelism{lock} ;  [...] ; deadlock-on .
```

The entry point of the program implementing the Dining Philosophers example is the function 'make in the class APPLICATION. The flag enabling the deadlock analysis is set to "on". This information is specified using the instruction settings('APPLICATION, 'make, false, deadlock-on).

A possible scenario that leads to a deadlock when running the above code is as follows. First, we initialize the left and right forks of the philosophers: p1 is assigned f1 and f2, re-

spectively, whereas `p2` is assigned `f2` and `f1`, respectively. Then, asynchronously, `p1` and `p2` (of *separate* type `PHILOSOPHER`) execute `eat_wrong`, which calls `pick_in_turn(left)`. In the context of `p1`, the actual value of `left` is `f1`, whereas for `p2` it is `f2`. Consequently, both resources `f1` and `f2`, respectively, might be locked "at the same time" by `p1` and `p2`, respectively. Note that `pick_in_turn` subsequently calls `pick_two` that, intuitively, should enable the philosophers to use both forks. Thus, if `f1` and `f2`, respectively, are locked by `p1` and `p2`, respectively, the calls `pick_two(f2, f1)` and `pick_two(f1, f2)` corresponding to `p1` and `p2` will (circularly) wait for each other to finish. According to the SCOOP semantics, `pick_two(f1, f2)` is waiting for `p2` to release `f2`, whereas `pick_two(f2, f1)` is waiting for `p1` to release `f1`, as the forks are passed to `pick_two(...)` as *separate* types. In the context of SCOOP, this corresponds to a Coffman deadlock [2].

We force the execution of the scenario above by applying the command/strategy `srew [...] using init ; parallelism{lock} ; [...] ; deadlock-on`. This determines Maude to first trigger the rule `[init]` in the SCOOP formalization. This makes all the required initializations of the *bootstrap* processor. Then, one of the processors that managed to *lock* the necessary resources is ("randomly") enabled to proceed to the asynchronous execution of its instruction stack, according to the strategy `parallelism{lock}`. The last step of the strategy calls the rule `[deadlock-on]` implementing the Coffman deadlock detection as in (1). (For a detailed description of SCOOP and its Maude formalization we refer the interested reader to the work in [8].)

We run the example by executing the command:
```
 > maude SCOOP.maude ..\examples\dining-philosophers-example.maude
```
The rewriting guided according to the aforementioned strategy leads to one solution identifying a Coffman deadlock. The relevant parts of the corresponding Maude output are as follows:

```
                \|||||||||||||||||||/
                --- Welcome to Maude ---
                /|||||||||||||||||||\
            Maude 2.6 built: Mar 31 2011 23:36:02
            Copyright 1997-2010 SRI International
                  Wed Sep 17 14:47:47 2014
[...]
===========================================
srewrite in SYSTEM : (import default __create__invariant_end(...) ;
    __create__invariant_end(...) ; __create__invariant_end(...) ;
    __create__invariant_end(...) ;) settings('APPLICATION, 'make, false,
    deadlock-on) using init ; parallelism{lock} ; [...] ; deadlock-on .

Solution 1
rewrites: 479677 in 2674887330ms cpu (7379ms real) (0 rewrites/second)
result Configuration: deadlock

No more solutions.
rewrites: 479677 in 2674887330ms cpu (7453ms real) (0 rewrites/second)
```